Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Incremental Dynamic Updates with First-class Contexts

Erwann Wernli[a] Mircea Lungu[a] Oscar Nierstrasz[a]

a. Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch

Abstract Highly available software systems occasionally need to be updated while avoiding downtime. Dynamic software updates reduce downtime, but still require the system to reach a quiescent state in which a global update can be performed. This can be difficult for multi-threaded systems. We present a novel approach to dynamic updates using first-class contexts, called *Theseus*. First-class contexts make global updates unnecessary: existing threads run to termination in an old context, while new threads start in a new, updated context; consistency between contexts is ensured with the help of bidirectional transformations. We show that for multi-threaded systems with coherent memory, first-class contexts offer a practical and flexible approach to dynamic updates, with acceptable overhead.

Keywords dynamic language; dynamic software update; reflection

1 Introduction

Real software systems must be regularly updated to keep up with changing requirements. Downtime may not be tolerable for highly available systems, which must then be updated dynamically, *e.g.*, web servers. The key challenge for dynamically updating such systems is to ensure consistency and correctness while maximizing availability.

The most popular scheme for dynamic updates is to interrupt the application to perform a global update of both the code and the state of the program [NHSO06, SHM09, GR83, HSD+12]. Such updates are inherently unsafe if performed at an arbitrary point in time: running threads might run both old and new code in an incoherent manner while old methods on the stack might presume type signatures that are no longer valid, possibly leading to run-time type errors. Quiescent global update points must be selected to ensure safe updates, but such points may be difficult to reach for multi-threaded systems [NH09, SHM09]. More generally, a global update might not be possible due to the nature of the change, for example it would fail to update anonymous connections to an FTP server that mandates authentication after the update: the missing information cannot be provided *a posteriori* [NHSO06].

Erwann Wernli, Mircea Lungu, Oscar Nierstrasz. *Incremental Dynamic Updates with First-class Contexts*. In Journal of Object Technology, vol. 12, no. 3, 2013, pages 1:1–27.
doi:10.5381/jot.2013.12.3.a1

Instead of global updates, we propose *incremental* updates. During an incremental update, clients might see different versions of the system until the update eventually completes. Each version is represented by a first-class *context*, which can be manipulated explicitly and enables the update scheme to be tailored to the nature of the application. For instance, the update of a web application can be rolled out on a per-thread, or per-session basis. In the latter case, visitors always see a consistent version of the application. Such a scheme would not be possible with a global update: one would need to wait until all existing sessions had expired before starting new ones. The overall consistency of the data is maintained by running bidirectional transformations to synchronize the representations of objects shared across contexts. We show that the number of such shared objects is significantly smaller than the number of objects local to a context, and that this strategy fits well with the nature of the event-based systems we are interested in.

Bidirectional transformations were used in prior work [CYH+11, Dug01, Cho09] to achieve dynamic updates, respectively for systems written in C [CYH+11], from a type-theoretic point of view [Dug01], and for databases [Cho09]. However, neither of these approaches modeled context explicitly, nor did they tackle object-oriented systems, taking into consideration type safety, performance, concurrency and garbage collection all together. This paper is an extended version of previous conference paper [Wer12]. It contains a stronger validation that assesses all phases of the update for a multi-threaded system with coherent memory. It has also been extended with sections detailing language constructs that were obstacles, to provide a more balanced view on the applicability of our approach. The main contribution of this paper is to demonstrate that first-class contexts offer a practical means to dynamically update software.

First, we present our *Theseus* approach informally with the help of a running example in section 2. We present our model in detail in section 3 and our implementation in section 4. We validate our approach in section 5 and demonstrate that it is practical. We put our approach into perspective in section 6 and we compare it with related work in section 7 before we conclude in section 8.

2 Running Example

To illustrate our approach let us consider the implementation of one of several available Smalltalk web servers[1]. Its architecture is simple; a web server listens to a port, and dispatches requests to so-called services that accept requests and produce responses. For the sake of our running example, let us assume that the server keeps count of the total number of requests that have been served. Figure 1 illustrates the relevant classes.

2.1 The Problem with Updates

Let us consider the evolution of the Response API, which introduces chunked data transfer[2], also depicted in Figure 1. Assume that instead of sending "Hello World" over the wire we need to produce a sensible answer that takes some time. Installing such an update *globally* raises several challenges. First, both the `HelloWorldService`

[1] See `http://www.squeaksource.com/WebClient.html` (The name is misleading since the project contains both an HTTP client *and* server)

[2] See version 75 of the project.

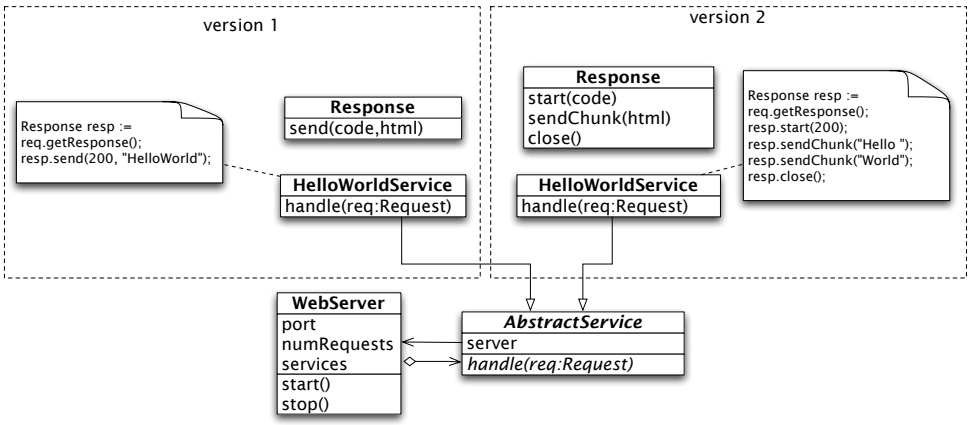

Figure 1 – Design of the web server and a simple behavioral update

and `Response` classes must be installed together: *How can we install multiple related classes atomically?*

Second, the methods impacted by the update can be modified or added only when no request is being served: *When can we guarantee that the installation will not interfere with the processing of ongoing requests?*

Rather than performing a global update, it would be more appealing to do an *incremental* update, where ongoing requests continue to be processed according to the old code, and new requests are served using the new code. Note that the granularity of the increment might differ depending on the update. We could imagine that the modification of a check-out process spanning multiple pages would imply that the increment be the web session rather than the web request. Our solution to enable incremental updates is to reify the execution *context* into a first-class entity.

Not only the *behavior* but also the *structure* of classes can change. Fields can be added or removed, and the type of a field can change. As a matter of fact, in a subsequent version of the project[3], the developers added a field `siteUrl` to the `WebServer` class. Unfortunately, the server is an object shared between multiple requests, and each service holds a reference back to the server. If the object structure is updated globally while different versions of the code run to serve requests, old versions of methods might access fields at the wrong index. While the problem for field addition can be solved easily by ensuring new fields are added at the end, we need to consider type changes as well. For instance, one could imagine that in the future newest versions will store the `siteUrl` as an `HttpUrl` rather than a `String`. Therefore, the general problem remains: *How can we ensure consistent access to objects whose structure (position or type of fields) has changed?*

Our solution to ensure consistent access is to keep one representation of the object per context and to synchronize the representations using bidirectional transformations. Once there is no reference any longer to a context, it is garbage-collected together with the corresponding representations of objects in that context.

[3]See version 82 of the project.

2.2 Lifecycle of an Incremental Update

Let us consider the addition of the field `siteUrl` in the `WebServer` class in more detail. The following steps describe how an *incremental* update can be installed with Theseus[4], an implementation of our approach, while avoiding the problems presented above.

First, the application must be adapted so that we can "push" an update to the system and activate it. Here is how one would typically adapt an event-based server system, such as a web server.

0. *Preparation.* First, a global variable `latestContext` is added to track the latest execution context to be used. Second, an administrative page is added to the web server where an administrator can push updates to the system; the uploaded code will be loaded dynamically. Third, the main loop that listens to incoming requests is modified so that when a new thread is spawned to handle the incoming request, the latest execution context is used. Fourth, the thread that listens to incoming connections in a loop is modified so that it is restarted periodically in the latest context. Note that the listening socket can be passed to the new thread without ever being closed.

After these preliminary modifications the system can be started, and now it supports dynamic updates. The life cycle of an update would be as follows:

1. *Bootstrap.* After the system bootstraps, the application runs in a default context named the *Root* context. The global variable `latestContext` is initialized to refer to the *Root* context. At this stage only one context exists and the system is similar to a non-contextual system.

2. *Offline evolution.* During development, the field `siteUrl` is added to `WebServer` and other related changes are installed.

3. *Update preparation.* The developer creates a class called `UpdatedContext`, which specifies the variations in the program to be rolled out dynamically. This is done by implementing a bidirectional transformation that converts the program state between the *Root* context and the *Updated* context. Objects will be transformed one at a time. By default, the identity transformation is assumed, and only a custom transformation for the `WebServer` class is necessary in our case.

4. *Update push.* Using the administrative web interface, the developer uploads the class `UpdatedContext` as well as the other classes that will be required by the context. The application loads the code dynamically. It detects that one class is a context and instantiates it. Contexts are related to each other by a ancestor-successor relationship. The ancestor of the newly created context is the active context at the time of code loading. The global variable `latestContext` is updated to refer to the newly created instance of the *Updated* context.

5. *Update activation.* When a new incoming request is accepted, the application spawns a new thread to serve the request in the `latestContext` (which is now the *Updated* context) while existing threads terminate in the *Root* context.

6. *Incremental update.* When the web server is accessed in the *Updated* context for the first time, the new version of the class is dynamically loaded, and the

[4]In reference to Theseus' paradox: if every part of a ship is replaced, is it still the same ship?

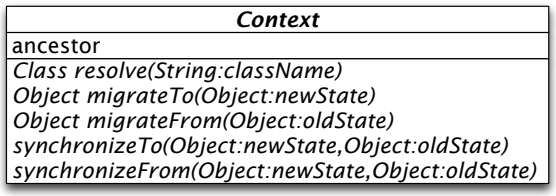

Figure 2 – The `Context` class.

instance is *migrated*. Migration is triggered when the object is accessed from a different context for the first time. In our case, this results in the fields `port` and `services` being copied, and the field `siteUrl` being initialized with a default value. Fields can be accessed safely from either the *Root* or *Updated* context, as each context has its own representation of the object. To ensure that the count of requests processed so far, `numRequests`, remains consistent in both contexts, bidirectional transformations between the representations are used. They are executed *lazily*: writing a new value in a field in one context only invalidates the representation of the object in the other context. The representation in the other context will be *synchronized* only when it is accessed again. Synchronization is performed lazily when changes happen to objects that have already been migrated.

7. *Garbage collection.* Eventually the listener thread is restarted, and all requests in the old context terminate. A context only holds weakly onto its ancestor so when no code runs in the old context any longer, the context is finalized. The finalization forces the migration of all objects in the old context that have not been migrated yet. The old context and its object representations can then be garbage-collected. It must be noted that at the conceptual level, all objects in memory are migrated. In practice, only objects that are shared between contexts need to be migrated.

3 First-class Context

Our approach relies on a simple, yet fundamental, language change: the state of an object is contextual. We assume throughout the rest of the paper that at most two contexts exist at a time, which we refer to as the "old" and "new" contexts. The model can be easily generalized to support any number of co-existing contexts, but the implementation would need to be revised since it takes advantage of this assumption.

3.1 User-defined Update Strategy

Contexts are first-class entities in our system. Programmers have complete control over the dynamic update of objects and classes. Contexts are ordinary instances of the class `Context`, shown in Figure 2. A context is responsible for maintaining the consistency of the representations of the objects belonging to it. Methods `Context.migrate{To|From}` and `Context.synchronize{To|From}` define the update strategy.

An object is initially *local* to the context in which it was created. When the object is first accessed in the "other" context (the context that isn't the one it was created

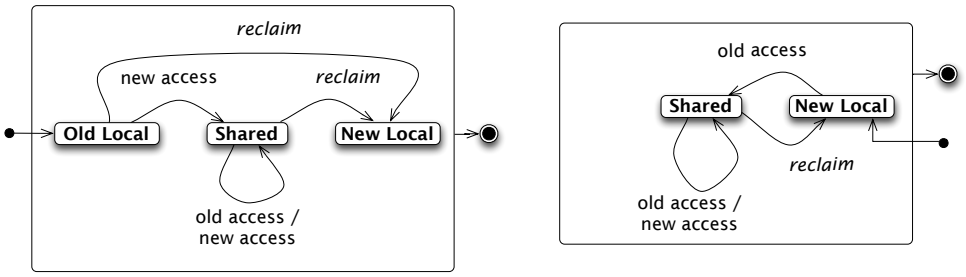

Figure 3 – Objects are originally local to a context. Depending on its access patterns, the object might become shared between contexts. Eventually, the object is reclaimed when the update completes and the object is local again.

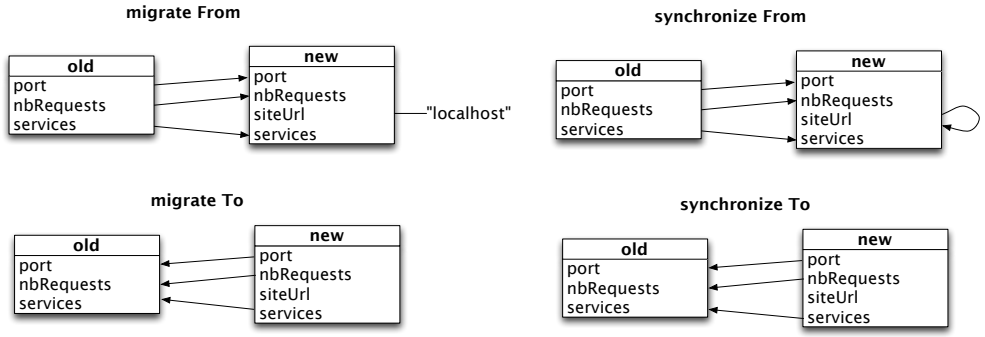

Figure 4 – The effects of the various methods that class `Context` mandates. Note that the arrow means a field copy operation and the method always applies to the new context.

in), the system triggers the *migration* of the object, after which the object is *shared* (see Figure 3). The migration creates the initial representation of the object in the "other" context. It is implemented in methods `Context.migrate{To|From}`. In our case, the migration of the web server from the old context to the new context would copy the existing fields *as is* and initialize the new field `siteUrl` with a predefined value (see Figure 4). By construction, either `migrateTo` or `migrateFrom` will be fired for an object, depending on whether it was originally created in the old or new context.

Methods `Context.synchronize{To|From}` are responsible for subsequent updates of the state when the object is shared. In the case of our example, the field `siteUrl` must not be initialized again. Both methods `synchronize{To|From}` can be fired multiple times for a given object depending on its access patterns from the old and new contexts. In the rest of the paper, we use the term "transformations" for both migrations and synchronizations.

Each context has an ancestor. Since the contexts are loaded dynamically in an unanticipated fashion the update strategy is encoded in the newest context and expressed in terms of its ancestor, never in terms of its successor: methods `Context.*From` use the old representation to update or create the new representation (new *from* old), and methods `Context.*To` use the new representation to create or update the old representation (old *to* new). The *Root* context is the only context that does not encode any transformation and has no ancestor.

3.2 Object Representations

Let us explain the semantics of our approach by considering how it could be implemented with a meta-circular interpreter. At the application level, an object has a unique identity and its state differs depending on the active context. An application thread will implicitly impact either the old or the new representation of an object, but cannot explicitly refer to either one. At the interpreter level, an application object is implemented by several representations. Transformations are reflective hooks that run at the interpreter level outside of any context. They can access simultaneously the old and new representations of an application object. From within a transformation, arbitrary messages cannot be sent to representations since the absence of context makes their behavior ill-defined. Representations must be manipulated with reflective state accesses that operate correctly. Listing 1 shows for instance the identity transformation.

```
Context>>synchronize: newState from: oldState
  "copy all instance variables one-to-one"
  newState instVarsDo: [ :idx |
    newState instVarAt: idx put: (oldState instVarAt: idx ) other
  ]
```

Listing 1 – Identity Transformation

The message `other` returns the representation in the "other" context. Certain objects in the system are primitive. They have a unique state in the system. This is notably the case for contexts, scalar values (string, numbers, *etc.*), and semaphores. Primitive objects can be used from the application and from the interpreter (*i.e.*, within transformations). Listing 2 shows a transformation between two states where the `address` and `domain` instance variables are concatenated. The transformation is allowed because strings are primitive.

```
| address domain |
address := newState instVar: #address.
domain := newState instVar: #domain.
"store the concatenation of address @ email in variable email"
oldState instVar: #email put: ( address , '@' , domain )
```

Listing 2 – Concatenation of two fields

Listing 3 shows the inverse transformation that splits the string into two parts. The method `split` returns an array, which isn't a primitive object. The manipulation of an array from within a transformation is possible with an explicit context switch which restores the existence of a context. Within a transformation, `self` represents the new context, and `self ancestor` the old context.

```
| address domain |
"perform a context switch"
self ancestor do: [
```

```
  | email |
  email := oldState instVar: #email.
  "extract the address and domain of the email"
  address := ( email split: '@' ) first.
  domain := ( email split: '@' ) second.
].
newState instVar: #address put: address.
newState instVar: #domain put: domain.
```

Listing 3 – Splitting a field into two

3.3 First-class Classes

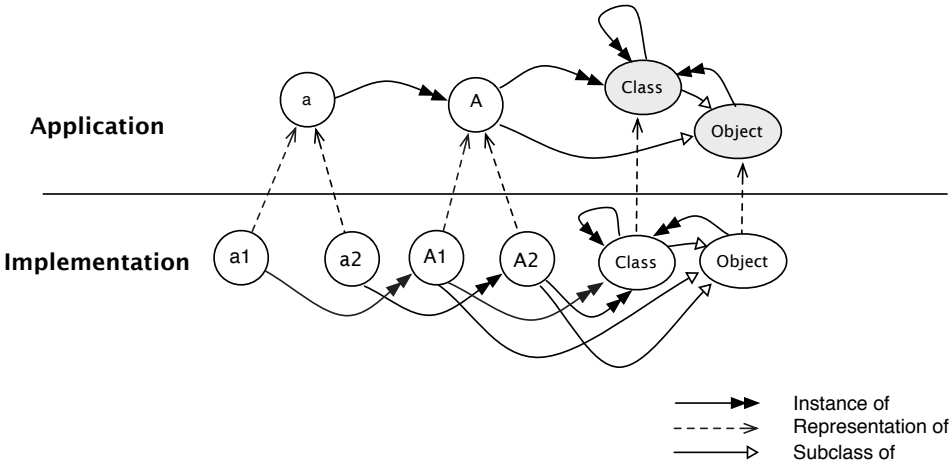

Figure 5 – Object `a` is an instance of class `A`. Both `a` and `A` are contextual objects since classes are first-class. At the interpreter level, contextual objects have multiple representations. Classes `Object` and `Class` are primitive and have a unique representation.

Classes are also objects in our approach. At the application level, a contextual object `a` is an instance of the contextual class `A`. At the interpreter level, contextual objects and classes have multiple representations, as depicted in Figure 5. A representation of a contextual object is an instance of the representation of the corresponding contextual class. For example, the object representation `a1` is an instance of the class representation `A1`.

When an object is migrated, its original representation is passed as a parameter to `migrate{To|From}`. The method creates and returns a second representation for the given context. For instance, the migration of `a1`, which is an instance of `A1`, results in `a2`, which is an instance of `A2`. The class can change only during migration. Indeed, methods `synchronize{From|to}` take as arguments two representations, but are not able to change the class they correspond to.

Classes are migrated similarly to regular objects. A specific representation of a class is passed as parameter to `migrate{To|From}`, which must return a second representation. For instance, the migration of `A1`, which is an instance of `Class`, returns `A2`, which is

also an instance of `Class`. Note that `Class` is a primitive in the system. Names of classes are literals in the source that resolve at run time to first-class classes. This binding is also contextual to support class renaming. Contexts are responsible for class name resolution and must implement the method `Context.resolve(String)`.

Similarly to reflective state accesses with `instVar:` and `instVar:put`, the method `class` can be used from within transformations to obtain the class of a representation. Methods `Context.migrate{To|From}` and `Context.synchronize{To|From}` can thus apply custom transformations for specific objects and default to the identity transformation for objects without structural changes. The code below applies a custom transformation to instances of the `Contact` class:

```
UpdatedContext>>synchronize: newState to: oldState
  ( newState class instVar: #name ) = #Contact ifTrue: [
    "custom transformation for instances of class Contact"
    | address domain |
    address := newState instVar: #address.
    domain := newState instVar: #domain.
    oldState instVar: #email put: ( address , '@' , domain ).
    ^ self.
  ]
  "default transformation (identity) for instances of other classes"
  ^ super synchronize: newState to: oldState.
```

Listing 4 – Custom transformation

3.4 Threads and Contexts

A thread can have one *active* context at a time. The runtime must be extended with a mechanism to query the active context, and to switch the current context. When a thread is forked, it will inherit the context of its parent thread. For convenience, a thread can be forked and change its context right after (initially it is the *Root* context). This way, code executed by the thread runs entirely in the given target context.

4 Implementation

We have implemented our approach in Pharo Smalltalk[5] using bytecode transformation to avoid changes to the virtual machine. A unique aspect of our implementation is that it does not rely on proxies or wrappers, which do not properly support self-reference, do not support adding or changing public method signature, and break reflection [PKS08, ORH02].

During an incremental update, a contextual object corresponds concretely to two objects in memory, one per context. To maintain the illusion that the old and new representations of an object have the same identity, we adapt the references when necessary: for instance, if `b1` is assigned to a field of `a1` in the old context, this results in `b2` being assigned to the corresponding field of `a2` in the newest context. Figure 6 depicts such a setting.

[5] `http://www.pharo-project.org`

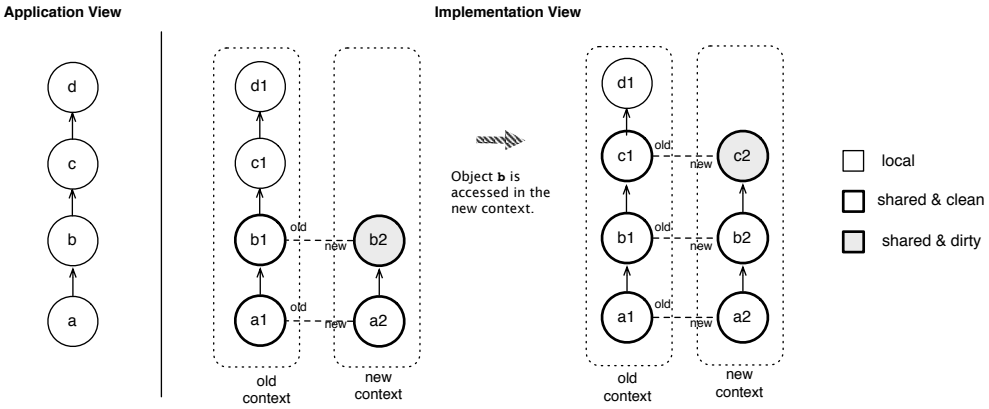

Figure 6 – From the application view, four contextual objects `a,b,c,d` and form a list. From the implementation view, shared objects have one representation per context, which can be either "clean" or "dirty". Objects are migrated lazily. When object `b` is accessed in the new context for the first time, the representation `b2` is synchronized. Since `b` refers to `c`, this triggers the migration of `c` and the representation `c2` is created, originally considered "dirty". An access to `c` in the new context would create the representation `d2`, *etc.* Dashed lines represent relationships visible only to the implementation, not the application.

Objects are migrated lazily, and can be either flagged as "clean" or "dirty". Dirty objects are out-of-date, and need to be synchronized upon the next access. Figure 6 shows the effect of an access to the dirty representation `b2`, which triggers the migration of the representation `c2` it references directly. After the synchronization, the two representations `b1` and `b2` of object `b` are clean. Subsequent writes to either representation would however result in the other one to be flagged as dirty. In the case of Figure 6, if `b2` is modified, `b1` would be marked as dirty.

We use bytecode rewriting to alter accesses to state and the way classes are resolved. Concretely, an extra check is added before each state read and state write to determine whether the object is shared between contexts. For state reads, if the object is shared and "dirty" it is first synchronized and then marked as "clean". Algorithm 1 shows in pseudo code what happens upon state reads in case of the identity transformation. For state writes, the new value is written and the other representation is invalidated and flagged as "dirty". We maintain the correspondence between representations using synthetic fields added during the program transformation.

We must ensure that all objects reachable from the old context have been migrated and are up-to-date before old representations are garbage-collected. Therefore, when the old context becomes eligible for garbage collection, the system traverses the object graph and forces the migration or synchronization of objects if necessary. This corresponds to the transitions labelled "reclaim" in Figure 3. In the case of Figure 6, the system would force the migration of `d1` before garbage collection. If the graph of reachable objects is big, the traversal can take long but can be conducted in the background with low priority.

```
1  if self is dirty and global then
   |  // synchronization
2  |  for field ← fields of self do
   |  |  // migration
3  |  |  if self.other.field is local then
4  |  |  |  migrate( self.other.field) ;
5  |  |  end
   |  |  // synchronization
6  |  |  self.field = self.other.field.other;
7  |  end
8  |  mark self as clean;
9  end
10 read field self.name;
```

Algorithm 1: Pseudo-code for state reads in the case of the identity transformation

4.1 Concurrency and Garbage Collection

We assume that the system has a coherent memory, as is the case for the Cog VM for Pharo. With a coherent memory, state reads and writes are atomic and side-effects are immediately visibile to all threads.

Since our implementation instruments state accesses with additional logic, it does not automatically preserve thread safety. A trivial way to preserve it would be to acquire a per-object lock for each state access. Migrations and synchronizations of an object would therefore never conflict. This would however lead to an unacceptable performance penalty.

Instead, we use a relaxed locking scheme where state accesses to objects that are local to a context do not require the acquisition of a lock. This scheme relies on the use of per-field dirty flags instead of per-object dirty flags. It also assumes that the original program is correctly synchronized and that reads and writes to a given field never happen concurrently. This should be the case for any program, since developers should never rely on the atomicity of state reads and writes even if memory is coherent.

Algorithm 2 adapts the pseudo code of the previous section to reflect this strategy. It assumes that two representations of a contextual object have the same lock, *i.e.*, self.lock = self.other.lock. An object is originally local to the context that created it. The object might later become shared between contexts. The object does not become shared at the moment it is accessed from another context, but when a reference to it is obtained (lines 7-13 in algorithm 2). For instance, in Figure 6, the old state `b1` is migrated when a new thread accesses `a2.f` and obtains a reference to `b2`. After migration, the migrated state is considered dirty. This corresponds to the mechanism of lazy propagation explained in the previous section.

Before we discuss the validity of our strategy, let us introduce some terminology: we use the term *old threads* for threads running in the old context, and *new threads* for threads running in the new context. Similarly, we use the terminology *old local object* and *new local object* for local objects created originally in the old or new context. The synthetic thread that forces the update of the reachable objects before garbage collection is referred to as the *background thread*. We assume that before it forces the update of an object, it acquires first its lock. As stated previously, concurrent reads and writes to the same field are excluded, since we assume the program is correctly

```
1  if self is local then
2  |  read field self.name;
3  else
4  |  acquire self.lock do
5  |  |  if self is not local then
6  |  |  |  if self.name is dirty then
   |  |  |  |  // migration
7  |  |  |  |  if self.other.name is local then
8  |  |  |  |  |  acquire self.other.name.lock do
9  |  |  |  |  |  |  if self.other.name is local then
10 |  |  |  |  |  |  |  migrate( self.other.name) ;
11 |  |  |  |  |  |  end
12 |  |  |  |  |  release
13 |  |  |  |  end
   |  |  |  |  // synchronization
14 |  |  |  |  self.name = self.other.name.other;
15 |  |  |  |  mark self.name as clean;
16 |  |  |  end
17 |  |  end
18 |  |  read field self.name;
19 |  release
20 end
```

Algorithm 2: Pseudo-code for state reads using a per-field dirty flag and locks for mutual exclusion.

synchronized. Let us use the variables f and g to refer to distinct fields of an object.

We can informally list all possible cases and show that the strategy effectively prevents *lost updates*:

Case 1: an old thread reads field f of an old local object. We consider three sub-cases: 1) Concurrent reads and writes to g by old threads. This doesn't lead to conflicts since fields are independent. 2) Concurrent reads and writes to g by new threads. This will trigger the migration of the object. After the migration, all fields of the new representation are considered dirty, except g. If it was a read, g holds the value of the old context; if it was a write it holds the updated value and the field in the old context is marked as dirty. In both cases, there is no conflict. 3) Forced updates by the background thread. By definition, this thread runs when there are no old threads any longer, so this case is not possible.

Case 2: an old thread writes into field f of an old local object. We consider three sub-cases: 1) Concurrent reads and writes to g by old threads. This doesn't lead to conflicts since fields are independent. 2) Concurrent reads and writes to g by new threads. This will trigger the migration of the object. If it is a read, all fields of the new representation except g will be dirty. The update to f is not lost and will be reflected if it is later accessed in the new context. The situation is identical for a write. 3) Forced updates by the background thread. By definition, this thread runs when there are no old threads any longer, so this case is not possible.

Case 3: old and new threads access an object shared between contexts. We consider two sub-cases: 1) The background thread isn't running. If the background thread is not running, there is no way for a shared object to become local again. All accesses will be mutually exclusive. 2) The background thread is running. If only new threads exist, the background thread can force the update of an object after which it will be local again. Lock acquisitions follow an idiom similar to double-checked locking [BBB+] where the condition *is local*/*is not local* is tested twice. If an object transitions from context-shared to context-local when a thread awaits for a lock, this change will be detected when the lock is acquired and won't lead to conflicts.

Case 4: a new thread reads field *f* of a new local object. We consider three sub-cases: 1) Concurrent reads and writes to g by new threads. This doesn't lead to conflicts since fields are independent. 2) Concurrent reads and writes to g by old threads. This will trigger the migration of the object. After the migration, all fields of the old representation are considered dirty, except g. If it was a read, g holds the value of the new context; if it was a write it holds the updated value and the field in the new context is marked as dirty. In both cases, there is no conflict. 3) Forced updates by the background thread. By definition, this thread does not mutate any data so it cannot lead to lost updates.

Case 5: a new thread writes into field *f* of a new local object. We consider three sub-cases: 1) Concurrent reads and writes to g by new threads. This doesn't lead to conflicts since fields are independent. 2) Concurrent reads and writes to g by old threads. This will trigger the migration of the object. If it is a read, all fields of the old representation expect g will be dirty. The update to f is not lost and will be reflected if it is later accessed in the old context. The situation is identical for a write. 3) Forced updates by the background thread. By definition, the background thread skips new local objects since they are by definition up-to-date. No update can be lost.

4.2 State Relocation

Transformations can be more complex than one-to-one mappings. For instance, instead of keeping track of the number of requests in `numRequests` using a primitive numeric type, the developer might introduce and use a class `Counter` for better encapsulation[6]. During the transformation, the actual count would be "relocated" from the web server object to the counter object that is now used. However, in this case, when the counter is incremented, the old representation of the web server with field `numRequests` needs to be invalidated. So far we have assumed that a write would invalidate only the representation of the object written to, which is not the case any longer. To support such transformations, the full interface enables custom invalidation on a per-field basis with `Context.invalidate{To|From}(Object oldState,Object newState,String field)`.

4.3 Special Language Constructs

The implementation described so far assumes a uniform language where state is accessible solely via instance variables, and object instantiation is realized with message sends to classes. Pharo Smalltalk is very close to this ideal language, with a few peculiarities nevertheless.

[6]This would be the refactoring "Replace Data Value with Object". See `http://www.refactoring.com`

Closures Closures are first-class in Smalltak. Closures are instances of `BlockClosure` and encode offsets of bytecode in the `CompiledMethod` they belong to. They are treated analogously to other objects. After migration, they reference the newest version of the corresponding `CompiledMethod`. Offsets are copied as is, assuming the same syntactic position of the closure in both versions of the sources. If this assumptions turns out wrong, it would be possible to write a custom transformation that corrects offsets during transformation.

Primitive Methods Our approach cannot intercept state changes from primitive methods. Primitive methods that operate on contextual objects must be adapted to work according to our model. Since primitive methods are specified with the `primitive` pragma, they can be renamed, and a wrapper method working correctly can be provided with the original name. The majority of primitive methods operate on primitive objects though, and do not need to be modified.

Cloning One special primitive method used pervasively is `copy` (and its variant `copyFrom:`). The correctly working wrapper must first ensure that if the object is shared, it is fully up-to-date. It can then be copied to produce a local clone.

Syntactic Sugar Pharo Smalltak has syntactic sugar for arrays `{...}`, literal arrays `#(...)`, and class variables that are visible to instances of a class and the class itself. These constructs are first desugared, then processed through our regular transformation.

Hashcode Comparing objects based on their identity works correctly with our approach: in Figure 6, code comparing `a == b` would consistently compare either `a1` with `b1`, or `a2` with `b2` depending on the context. The hash code of `a` would however be different depending on the context, breaking notably the collection classes that use hash codes to position elements in internal data structures. To ensure that different versions of an object have the same hash code, we keep a unique identity in an additional synthetic field.

Continuations The stack is a sequence of activation frames. Continuations can capture context switches or ignore them. In the first case, the continuation would be a primitive object that would not be modified when exchanged across contexts; calling a continuation would restore the corresponding context switches along the way. In the second case, the continuation would be contextual and its corresponding activation frames would be adjusted when exchanged across contexts. Activation frames could be migrated similarly to closures, assuming that methods on the stack have not been modified between versions. If activated methods would have been modified, the adjustment of the continuation would require a mapping that might be difficult to achieve, as shown by Makris and Bazzi [MB09]. We have not implemented support for continuations.

Exceptions In our approach, threads run consistently in one context (see subsection 3.4). An exception is an object that is thrown and caught within the same context. If a thread can switch temporarily its context, objects flowing in and out of the boundary of the temporary context must be migrated accordingly, which includes exceptions. The migration of the stack frames the exception refers to would lead in this case to issues similar to those with continuations.

4.4 Further Details

We used a custom compiler to rewrite the bytecode of contextual classes. We instrumented copies of kernel classes (array, dictionary, etc.) to avoid metacircularity issues during development. Also, the `Object` class cannot be extended with the necessary information for our model (namely the synthetic fields to related versions with each other) and we instead extended the subclasses of `Object`. Primitive classes (see subsection 3.2) do not require any bytecode rewriting. This design requires methods of primitive classes like `String»split` to be trapped: invoked from the environment, an instance of the uninstrumented collection class is returned; invoked from our application, an instance of the instrumented collection class is returned. The complete set of such methods has not been identified and trapped, which can lead to minor bugs.

The active context is stored in a thread-local variable and we add a new method to fork a closure in a specific context, *e.g.*,`[ ... ] forkInContext: aContext`. When a closure is forked, it becomes a shared contextual object and is migrated. As the program proceeds, objects referenced by the closure are migrated lazily when accessed. Contexts hold only weak references to their ancestor and implement the method `Object»#finalize`, which forces the migration of all reachable objects before the context becomes eligible for garbage collection.

5 Validation

5.1 Evolution

We conducted a first experiment whose goal was to assess whether our model could support long-term evolution, that is, whether it could sustain successive updates. We considered the small web server of section 2, which despite its simplicity cannot be updated easily with global updates. The sever has a simple architecture and is comprised of 7 classes: `WebServer`, `WebRequest`, `WebResponse`, `WebMessage`, `WebUtils`, `WebCookie`, `WebSocket`. We selected the 4 last versions with effective changes: version 75 introduced chunked data transfer, version 78 fixed a bug in the encoding of URL, version 82 introduced `siteUrl`, and version 84 fixed a bug in MIME multipart support.

To restart periodically, the listener thread executing `WebServer»runListener` was adapted to accept incoming connections only during 1 second, after which a new listener thread is restarted (see Listing 5). It required only a couple of lines to be changed. The method `WebServer»asyncHandleConnectionFrom:` spawns a new thread per connection. It was simply modified to spawn the thread in the latest context.

```
WebServer>>runListener
  | connectionSocket startTime |
  startTime := Time now.
  [ (Time millisecondsSince: startTime ) < 1000 ] whileTrue: [
    connectionSocket := listenerSocket waitForAcceptFor: 5.
    self asyncHandleConnectionFrom: connectionSocket.
  ].
  "the listener restarts itself in the latest context"
  [ self runListener ] forkInContext: CurrentContext lastest
```

Listing 5 – Modified listener thread. It omits error handling code for readability.

We implemented the hello world service of section 2. Only one update required us to write a custom transformation: the one that introduced the `siteUrl` field, which we initialized to a default value.

```
UpdatedContext>>migrateFrom: oldState
   ( oldState class instVar: #name ) = #WebServer ifTrue: [
    | newState |
    newState := oldState class other new.
    newState instVar: #port put: (oldState instVar: #port).
    newState instVar: #nbReq put: (oldState instVar: #nbReq)
    newState instVar: #services put: (oldState instVar: #services)
    "initialize the new field"
    newState instVar: #siteUrl put: 'http//localhost'.
    ^ newState.
  ]
  ^ super migrateFrom: oldState.
```

Listing 6 – Migration that initializes field `siteUrl`

We ran the 4 successive dynamic updates, and verified that once it was no longer used, the old context would be garbage-collected. Since web browsers keep one connection alive for multiple requests, we could observe different versions of the services in two browsers.

5.2 Run-time Characteristics

For the second experiment, we picked a typical technology stack with two well-known production projects: the Swazoo[7] web server and the Seaside[8] web framework. This corresponds to several hundred classes. Similarly to the previous experiment, the web server was adapted to periodically restart its listener thread and process requests in the latest context. We were interested in the run-time characteristics and in assessing (1) whether our assumptions about object sharing hold, (2) what is the memory overhead, and (3) what is the time overhead.

As a case study, we considered the counter component example that comes with the Seaside distribution. During maintenance, only few classes change. Most objects are migrated with the identity transformation, and only certain objects require custom transformations. The exact nature of the transformation is not significant. Therefore, for the sake of simplicity, we artificially updated the system and used the identity transformation for all objects.

We were interested to assess the overhead of our implementation during the five following phases:

(i) with only the old context when no object is shared,

(ii) during the incremental update when objects are shared and migrated lazily,

(iii) after objects have been migrated but are still considered shared,

[7] `http://www.swazoo.org`
[8] `http://www.seaside.st`

Table 1 – Read/write ratios and heap size per phase. The star (*) indicates phases with increasing memory consumption, for which we considered the peak.

	i	ii* / iii	iv*	v
# reads	59'908	60'952	62'021	61'568
# shared reads	0	29'278	29'666	0
% shared	0.00	46.80	46.63	0.00
# writes	3'773	3'749	3'777	3'745
# shared writes	0	174	174	0
% shared	0.00	4.64	4.61	0.00
# objects	48'905	49'390	49'829	49'961
# shared objects	0	1'118	36'591	0
% shared	0.00	2.26	73.43	0.00
Heap size	1'063'492	1'078'494	1'087'065	1'085'704
Shared heap size	0	30'160	858'695	0
% shared	0.00	2.80	78.99	0.00

(iv) when the old context is finalized and the system forces the migration/update of all objects reachable in the old context, and

(v) after the old context has been garbage-collected and the system runs as in (i).

5.2.1 Object Sharing and Memory Overhead

We studied object sharing and memory overhead by manually incrementing the counter from one browser session. We tracked the number of reads and writes to objects shared between contexts, and to objects local to a context. To account for "sharable" objects in the heap, we tracked all objects reachable from classes and all objects reachable from variables captured by forked closures. To account for the "local" objects in the heap, we tracked all objects that were receivers or return values of message sends.

We measured first the memory after 5 increments of the counter, before any update. This corresponds to phase (i). We reset the tracking, installed the update, incremented the counter 5 more times and then measured memory again. This corresponds to phase (iii) which itself corresponds to the peak of memory of phase (ii). We then measured the memory at the peak of consumption for phase (iv), when the complete graph of objects has been traversed but no object has been reclaimed yet. Finally, we reset the tracking, incremented the counter 5 times, and measured again the memory. This corresponds to phase (v), after memory has been reclaimed.

Since our implementation uses a copy of the kernel classes, we observe only the effects of our application in isolation from the Smalltalk environment. We track receivers and return values at call sites, which correctly considers primitive objects whose classes haven't been instrumented (see section 4). Since transient objects are tracked, they will not be garbage-collected. Our approach does not measure the effective size of the heap, but estimates the upper bound. This upper bound is thus the sum of the sizes of the individual objects. The size of an object was computed with

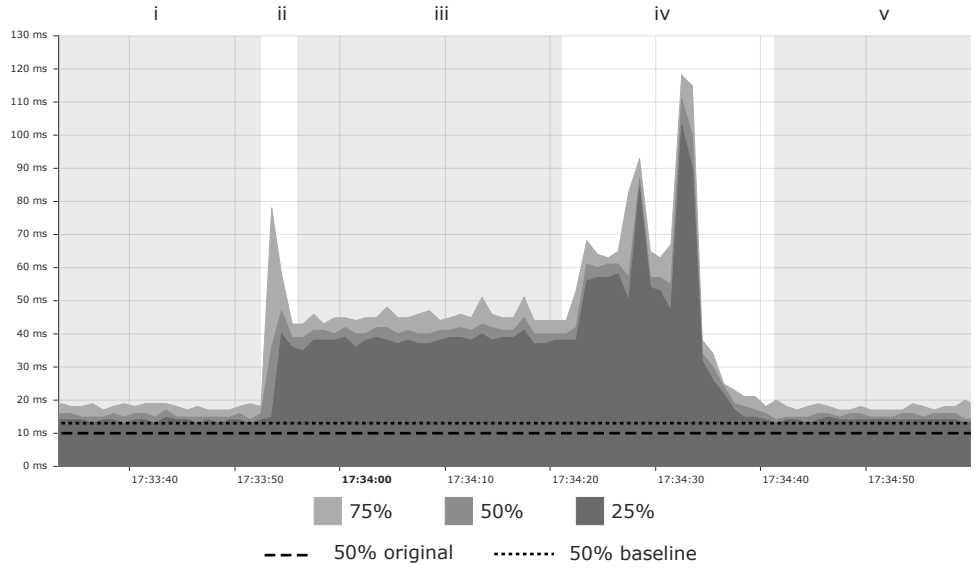

Figure 7 – Response time quantiles for a typical run of the load test. Quantiles are computed using measures within a window of 1 second. The various phases (i) to (v) of the dynamic update are highlighted with different background colors.

a known algorithm that considers its structure[9], then it was doubled in the case the object is shared. When an object is shared between contexts, we need to keep for each object representation the following information: a semaphore (see subsection 4.1), a unique identifier (see subsection 4.3), dirty flags (see subsection 4.1), and one reference to the "other" representation (see subsection 4.1). Table 1 does not account for this overhead that is very implementation-specific.

We make the following observations from the results presented in Table 1:

- There are an order of magnitude more reads than writes.
- The percentage of objects effectively shared (2.26%) during phase (iii) is smaller than the percentage of objects that are shareable (73.43%) and thus migrated in phase (iv).
- Shared reads represent 46% of all reads. Shared writes are mostly neglectable (4.6% in phases (iii) and (iv)). This supports the idea that an overhead for accesses to shared objects is tolerable.
- The heap is composed mostly of shareable objects (73.43%, phase (iv)) and only few transient, local objects. This might be related to the stateful nature of the Seaside framework. This means however that our approach might entail an important memory overhead in the case study.

5.2.2 Time Overhead

We studied the time overhead in a multi-threaded scenario. We conducted load tests[10] with the following setting: 10 concurrent visitors connect to the website and start a

[9] See Seaside's internal memory profiler, and the method `WAMemoryItem»sizeOfObject`

[10] `http://jmeter.apache.org`

web session each. They go to the counter page. From there, they keep incrementing the counter as quickly as possible. The system had been modified to install updates per request. The server and the load testing tool ran both on the same machine (a 2.3 GHz Intel Core i7 laptop), to minimize the effect of the network. We ran the server on the CogVM 6.0.

The results for the time overhead are presented in Figure 7, which shows the quantiles in response time for the various phases. The dashed and dotted lines represent the median of the response time for the original system, and the baseline system. The baseline system is the original system with one level of indirection for state accesses that go through synthetic accessors. This helps one to compare the overhead of the approach instead of the cost of the indirection mechanism, which could be aggressively inlined if necessary. We make the following observations from the results of Figure 7:

- The sole use of synthetic getters and setters (without additional logic) entails a 30% overhead, as can be seen in the difference between the original system and the baseline (10ms vs 13ms). Other virtual machines might be able to aggressively optimize this case. The overhead between the baseline and phases (i) and (iv) is 15% (13ms vs. 15ms), supporting the idea that our approach is attractive at "steady state".
- The five phases are clearly visible. Phase (ii) corresponds to a peak when the majority of objects are first migrated. Phase (iii) corresponds to a plateau where performance is degraded due to lock acquisition, checks for "dirtiness" of object representations, and resulting synchronizations. Phase (iv) shows further degradation when the system operates in background the synchronization of all objects reachable in the new context. Eventually, the system reverts back to its original performance (iv).
- The performance degradation between phases (i) and (iii) is of factor 2.6, which is tolerable for a short period of time.
- Giving a lower priority to the background task during phase (iv) would make the update take longer to complete, but lower the performance degradation.

Overall, the benchmark shows the expected profile, and suggest our approach can be made practical for realistic production systems.

6 Discussion

We discuss in this section the applicability of our approach and its implementation from three different perspectives:

6.1 Portability

Our approach and implementation technique are portable to other object-oriented languages. The approach can notably be ported to a statically typed language. Particular language constructs of the target language might however pose obstacles, specifically those we listed in subsection 4.3. Experiments porting the approach to Java showed it is feasible [WLN12] with the following language constructs as obstacles: constructors, nested classes, arrays, concurrency control in the language semantics.

Our locking scheme resembles the double-checked locking [BBB+] idiom which is correct only for systems with a coherent memory. Java has for instance a relaxed memory model (JSR-133) where the double-checked locking is correct only if the field that is tested is declared `volatile` [BBB+]. Our locking scheme would need to be revised accordingly for a full port to Java.

6.2 Performance

A drawback of our implementation is that shared objects need two representations, even if they are structurally identical and will use the identity transformation. Wrappers would make it possible to keep only one representation in such cases, but pose problems of self-reference, do not support adding or changing method signatures, and break reflection [PKS08, ORH02, PC12]. The benefit of our implementation is that it avoids such problems. Our implementation entails a performance degradation due to internal locking when shared objects are accessed. These locks are however typically uncontended. Their impact on performance depends on how well uncontended locking is optimized by the virtual machine. Two directions could be explored to reduce locking: 1) synchronize groups of fields at precise locations, instead of each individual field (*e.g.*, synchronize all fields a method uses at once at the beginning and end of the method), and 2) emulate safepoints and run the pre-garbage collection thread exclusively from application threads, making migration the only operation that requires locking.

6.3 Development Effort

The impact on development is small. Developers must modify the application's thread management to make the application updatable. Essentially, long-running loops must be modified to restart periodically and requests must be processed in the desired context. Threads executing long-running loops can usually be aborted and restarted without problem. Adapting request dispatching is also usually easy and entails only few local changes. In our first experiment, these changes represent about 10 lines of code. The use of thread pools might complicate the adaptation of the application in which case the pool must be adapted to renew its threads periodically [WLN12]. Reflective code doesn't need to be adapted since object representations are really instances of their respective classes and reflective code works correctly.

Writing transformations to transfer the application state requires additional effort. Compared to other dynamic update mechanisms, there must exist a state mapping only for shared entities (not all entities), but the mapping must be bidirectional (not unidirectional). Transformations are usually simple (field addition, renaming, suppression, type conversion) and complex transformations are very occasional [SHM09, MPG+00, BLS+03]. In our first experiment, only one update required a transformation, which was simple. Recent works showed that static [POMS09] and dynamic analysis [MHSM12] can generate most of the needed transformations automatically. It would be interesting to assess whether we can extend these techniques for bidirectional transformations as well.

7 Related Work

A common technique to achieve hot updates is to use redundant hardware [HN05], possibly using "session affinity" to ensure that the traffic of a given client is always routed to the same server. Our approach is more lightweight and enables the migration of the state shared across contexts, notably persistent objects. Also, an advantage of being reflective is that the software can "patch itself" as soon as patches become available.

A large body of research has tackled the dynamic update of applications. The main challenge is to reconcile safety and practicality. Systems that support immediate code changes are very practical but subject to limitations or safety issues. Dynamic languages, including Smalltalk, belong to this category. If an object attempts to invoke a method that was suppressed, an error is raised. Several approaches of this kind have also been devised for Java [Dmi01, ORH02, KV12, GJ09, CPG11, WWS11, PKC+11], with various levels of flexibility (a good comparison can be found in [GJ09]). To be type-safe, HotSwap [Dmi01] imposes restrictions and only method bodies can be updated. The most recent approaches (JavAdaptor [PKC+12], DCEVM [WWS11], Javaleon [GJ09]) overcame most of these restrictions, and provide a similar flexibility as dynamic languages. Some approaches use bytecode transformation (JavAdaptor [PKC+12], Javaleon [GJ09]) or custom virtual machines (DCEVM [WWS11]).

Systems that impose constraints on the timing of updates are safe, but less practical since temporal *update points* must first be identified. Such systems have been devised for C (Ginseng [HN05, NHSO06], UpStare [MB09], Kitsune [HSD+12]), and Java (JVolve [SHM09], DVM [MPG+00]). Update points might be hard to reach, especially in multi-threaded applications [NH09, SHM09], and this compromises the timely installation of updates. Our approach entails the identification of *context switch* points, but relaxes the need for threads to reach the points simultaneously.

Some mechanisms diverge from a global update and enable different versions of the code or entities to coexist. In the most simple scheme, old entities are simply not migrated at all and only new entities use the updated type definition [HG98], or this burden might be left to the developer who must request the migration explicitly [Gem07]. The granularity of the update for such approaches is the object; it is hard to guarantee *version consistency* and to ensure that mutually compatible versions of objects will always be used. When leveraged, transactions [BLS+03, PC12] provide version consistency but impede mutations of shared entities. Contexts enable mutations of shared entities and can be long-lived, thanks to the use of bidirectional transformations. With asynchronous communication between objects, the update of an object can wait until dependent objects have been upgraded in order to remain type-safe [JKY09].

To the best of our knowledge, only three approaches rely on bidirectional transformations to ease dynamic updates. POLUS is a dynamic updating system for C [CYH+11] which maintains coherence between versions by running synchronizations on writes. We synchronize lazily on read, operate at the level of objects, and take garbage collection into account. Duggan [Dug01] formalized a type system that adapts objects back and forth: when the run-time version tag of an object doesn't match the version expected statically, the system converts the object with an adapter. We do not rely on static typing but on dynamic scoping with first-class contexts, we address garbage collection, concurrency, and provide a working implementation. Oracle enables a table to have two versions that are kept consistent thanks to bidirectional "cross-edition triggers" [Cho09].

Schema evolution addresses the update of persistent object stores, which closely relates to dynamic updates. To cope with the volume of data, migrations should happen lazily. To be type-safe, objects should be migrated in a valid order (*e.g.*, points of a rectangle must be migrated before the rectangle itself) [BLS+03, PC12]. Our approach migrates objects lazily, and avoids the problem of ordering by keeping both versions as long as necessary.

Class loaders [LB98] allow classes to be loaded dynamically in Java. Types seen within a class loader never change, which ensures type safety and version consistency, similarly to our notion of context. Two versions of a class loaded by two different class loaders are different types, which makes sharing objects between class loaders complicated. This is unlike our approach which supports the migration of classes and objects between contexts.

Context-oriented programming [HCN08] enables fine-grained variations based on dynamic attributes, *e.g.*, dynamically activated "layers". It focuses on behavioral changes with multi-dimensional dispatch, and does not address changing the structure and state of objects as is necessary for dynamic updates. There exist many mechanisms to scope changes statically, *e.g.*, Classboxes [Ber05], but they are not used to adapt software at run time.

How to keep two corresponding data structures synchronized is related both to the view-update problem [Kel85] and lenses [BPV06, FPP08, HPW11, WGW11]. We need in our approach to define a pair of transformations to map the source to the view, and the view to the source. Lenses are bidirectional programs that specify both a view definition and update policy. Lenses can be state-based or operation-based. State-based lenses synchronize structures as a whole, without knowing where the changes occurred, whereas operation-based lenses propagate local changes (or edits). Using lenses to express transformations would make their expression more compact and less error-prone. Since we synchronize lazily, edits are lost and we would need to use state-based lenses.

8 Conclusion

Existing approaches to dynamically update software systems entail trade-offs in terms of safety, practicality, and timeliness. We propose a novel, incremental approach to dynamic software updates. During an incremental update, clients might see different versions of the system, which avoids the need for the system to reach a quiescent, global update point.

Each version of the system is reified into a first-class context. Existing objects are gradually migrated to the new context, and objects that are shared between old and new contexts are kept consistent with the help of bidirectional transformations. We have implemented our approach in a dynamic language with a strong memory model and conducted experiments on two existing web servers. Results indicate that only a fraction of accesses concern objects shared between contexts which makes the cost of bidirectional transformations tolerable. Only few modifications to the original web servers were required to make them updatable, and simple transformations can be easily expressed in our approach.

This work opens up several research directions: exploring different granularity of increments, further improving the performance, and adapting the design for weak memory models.

References

[BBB+] David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, Jeremy Manson, John D. Mitchell, Kelvin Nilsen, Bill Pugh, and Emin Gun Sirer. The "double-checked locking is broken" declaration.

[Ber05] Alexandre Bergel. *Classboxes — Controlling Visibility of Class Extensions*. PhD thesis, University of Bern, November 2005. Available from: `http://scg.unibe.ch/archive/phd/bergel-phd.pdf`.

[BLS+03] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, 38(11):403–417, 2003. Available from: `10.1145/949343.949341`, `doi:10.1145/949343.949341`.

[BPV06] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '06, pages 338–347, New York, NY, USA, 2006. ACM. Available from: `10.1145/1142351.1142399`, `doi:10.1145/1142351.1142399`.

[Cho09] Alan Choi. Online application upgrade using edition-based redefinition. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 4:1–4:5, New York, NY, USA, 2009. ACM. `doi:10.1145/1656437.1656443`.

[CPG11] Susanne Cech Previtali and Thomas R. Gross. Aspect-based dynamic software updating: a model and its empirical evaluation. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 105–116, New York, NY, USA, 2011. ACM. `doi:10.1145/1960275.1960289`.

[CYH+11] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang, and Pen-Chung Yew. Dynamic software updating using a relaxed consistency model. *IEEE Trans. Softw. Eng.*, 37(5):679–694, September 2011. Available from: `http://dx.doi.org/10.1109/TSE.2010.79`, `doi:10.1109/TSE.2010.79`.

[Dmi01] M. Dmitriev. Towards flexible and safe technology for runtime evolution of Java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001*, October 2001.

[Dug01] Dominic Duggan. Type-based hot swapping of running modules (extended abstract). In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 62–73, New York, NY, USA, 2001. ACM. Available from: `http://doi.acm.org/10.1145/507635.507645`, `doi:10.1145/507635.507645`.

[FPP08] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 383–396, New York, NY, USA, 2008. ACM. `doi:10.1145/1411204.1411257`.

[Gem07] Gemstone/s programming guide, 2007. Available from: `http://seaside.gemstone.com/docs/GS64-ProgGuide-2.2.pdf`.

[GJ09] Allan Raundahl Gregersen and Bo Norregaard Jorgensen. Dynamic update of Java applications — balancing change flexibility vs programming transparency. *J. Softw. Maint. Evol.*, 21:81–112, mar 2009. Available from: http://portal.acm.org/citation.cfm?id=1526497.1526501, doi:10.1002/smr.v21:2.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983. Available from: http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

[HCN08] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008. Available from: http://www.jot.fm/contents/issue_2008_03/article4.htmlhttp://www.jot.fm/issues/issue_2008_03/article4.pdf, doi:10.5381/jot.2008.7.3.a4.

[HG98] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes: a lightweight mechanism to update code in a running program. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 6–6, Berkeley, CA, USA, 1998. USENIX Association. Available from: http://portal.acm.org/citation.cfm?id=1268256.1268262.

[HN05] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, nov 2005. doi:10.1145/1108970.1108971.

[HPW11] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric lenses. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 371–384, New York, NY, USA, 2011. ACM. Available from: 10.1145/1926385.1926428, doi:10.1145/1926385.1926428.

[HSD+12] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune: efficient, general-purpose dynamic software updating for c, 2012. Available from: http://doi.acm.org/10.1145/2384616.2384635, doi:10.1145/2384616.2384635.

[JKY09] Einar Broch Johnsen, Marcel Kyas, and Ingrid Chieh Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 596–611, Berlin, Heidelberg, 2009. Springer-Verlag. Available from: http://dx.doi.org/10.1007/978-3-642-05089-3_38, doi:/10.1007/978-3-642-05089-3_38.

[Kel85] Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '85, pages 154–163, New York, NY, USA, 1985. ACM. Available from: http://doi.acm.org/10.1145/325405.325423, doi:10.1145/325405.325423.

[KV12] Jevgeni Kabanov and Varmo Vene. A thousand years of productivity: the jrebel story. *Software: Practice and Experience*, pages n/a–n/a, 2012. Available from: http://dx.doi.org/10.1002/spe.2158, doi:10.1002/spe.2158.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Proceedings of OOPSLA '98, ACM SIGPLAN Notices*, pages 36–44, 1998. doi:10.1145/286936.286945.

[MB09] Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 31–31, Berkeley, CA, USA, 2009. USENIX Association. Available from: http://portal.acm.org/citation.cfm?id=1855807.1855838.

[MHSM12] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. Automating object transformations for dynamic software updating. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 265–280, New York, NY, USA, 2012. ACM. Available from: http://doi.acm.org/10.1145/2384616.2384636, doi:10.1145/2384616.2384636.

[MPG+00] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361. Springer-Verlag, 2000. doi:10.1007/3-540-45102-1_17.

[NH09] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 13–24, New York, NY, USA, 2009. ACM. doi:10.1145/1543135.1542479.

[NHSO06] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 72–83, New York, NY, USA, 2006. ACM. Available from: http://doi.acm.org/10.1145/1133981.1133991, doi:10.1145/1133981.1133991.

[ORH02] Alessandro Orso, Anup Rao, and Mary Jean Harrold. A Technique for Dynamic Updating of Java Software. *Software Maintenance, IEEE International Conference on*, 0:0649+, 2002. Available from: http://dx.doi.org/10.1109/ICSM.2002.1167829, doi:10.1109/ICSM.2002.1167829.

[PC12] Luís Pina and João P. Cachopo. Atomic dynamic upgrades using software transactional memory. In *HotSWUp*, pages 21–25, 2012. doi:dx.doi.org/10.1109/HotSWUp.2012.6226612.

[PKC+11] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. Flexible dynamic software updates of java applications: Tool support and case study. Technical Report 04, School of Computer Science, University of Magdeburg, 2011. Available from: http://wwwiti.cs.uni-magdeburg.de/iti_db/publikationen/ps/auto/FIN-04-2011.pdf.

[PKC+12] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. JavAdaptor—flexible runtime updates of Java applications. *Software: Practice and*

Experience, pages n/a–n/a, 2012. Available from: http://dx.doi.org/10.1002/spe.2107, doi:10.1002/spe.2107.

[PKS08] Mario Pukall, Christian Kästner, and Gunter Saake. Towards unanticipated runtime adaptation of java applications. In *APSEC '08: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, pages 85–92, Washington, DC, USA, 2008. IEEE Computer Society. doi:10.1109/APSEC.2008.66.

[POMS09] Marco Piccioni, Manuel Oriol, Bertrand Meyer, and Teseo Schneider. An ide-based, integrated solution to schema evolution of object-oriented software. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 650–654, Washington, DC, USA, 2009. IEEE Computer Society. Available from: http://dx.doi.org/10.1109/ASE.2009.100, doi:10.1109/ASE.2009.100.

[SHM09] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: a VM-centric approach. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 1–12, New York, NY, USA, 2009. ACM. Available from: http://doi.acm.org/10.1145/1542476.1542478, doi:10.1145/1542476.1542478.

[Wer12] Erwann Wernli. Theseus: Whole updates of Java server applications. In *Proceedings of HotSWUp 2012 (Fourth Workshop on Hot Topics in Software Upgrades)*, pages 41–45, June 2012. Available from: http://scg.unibe.ch/archive/papers/Wern12b.pdf, doi:10.1109/HotSWUp.2012.6226616.

[WGW11] Meng Wang, Jeremy Gibbons, and Nicolas Wu. Incremental updates for efficient bidirectional transformations. *SIGPLAN Not.*, 46:392–403, sep 2011. Available from: 10.1145/2034574.2034825, doi:10.1145/2034574.2034825.

[WLN12] Erwann Wernli, Mircea Lungu, and Oscar Nierstrasz. Incremental dynamic updates with first-class contexts. In *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2012*, pages 304–319, 2012. Available from: http://scg.unibe.ch/archive/papers/Wern12a.pdf, doi:10.1007/978-3-642-30561-0_21.

[WWS11] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Unrestricted and safe dynamic code evolution for Java. *Science of Computer Programming*, July 2011. Available from: http://dx.doi.org/10.1016/j.scico.2011.06.005, doi:10.1016/j.scico.2011.06.005.

About the authors

Erwann Wernli is a PhD student at SCG, where he conducts research into run-time adaptations and object-oriented languages. Visit him at `http://scg.unibe.ch/staff/ewernli`.

Mircea Lungu is a PhD researcher at the Institute of Computer Science (IAM) of the University of Bern. His interests range from software evolution analysis to programming languages and mobile computing. Visit him at `http://scg.unibe.ch/staff/mircea`

Oscar Nierstrasz is a professor of computer science at the Institute of Computer Science (IAM) of the University of Bern, where he founded the Software Composition Group in 1994. See: `http://scg.unibe.ch/oscar`.

Acknowledgments We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 - Sept. 2012) and "Agile Software Assessment" (SNSF project Np. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).